Module 1 Lab — Session 1: The Data Model

Value

M1 — C# 14 Essentials

1 (Null Safety), 2 (Primitives), 3 (Encapsulation — all parts), 3B (Interface Contracts)

Tier 1 (LO 1.1, 1.2) + Tier 2 (LO 1.3, 1.4)

Welcome to the TMS Development Team

Assume you are the newest backend engineer on the Training Management System the platform that tracks students, manages course enrollment, calculates grades, and allocates government training grants for CTBE.

The lead architect has handed you a stack of incident reports from the legacy system. The Ministry of Education's latest audit found a phantom `0.03 Birr` balance distributed across 100,000 student accounts a floating-point bug that survived three years of testing. A midnight batch job crashed with **`NullReferenceException`** because nobody checked whether a student's region field was empty. The enrollment confirmation pipeline silently overwrote course codes because the data objects had public setters. And on the day final exam grades were published, the server froze not from CPU or memory pressure, but because every database call blocked the thread pool with .Result.

Your job across these three sessions is to rebuild the core logic engine from scratch, fixing each category of defect the right way. By the time you finish, your code will be typesafe, financially precise, immutable where it should be, `queryable` with LINQ, and non blocking under load.

Before You Begin — Create the Project

Every exercise in this workbook runs inside a single .NET 10 console application. Create it once now. Do not skip this step.

Open your terminal and run these commands one at a time:

```
# Check your SDK version — must show 10.x
dotnet --version

# Create the Console Application
dotnet new console -n TmsCore --framework net10.0

# Move into the project directory
```

```
cd TmsCore

# Open it in VS Code (or Visual Studio)
code .
```

Once the project opens, find TmsCore.csproj. It should look like this:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

Three settings matter:

- **TargetFramework net10.0** — you are running the latest Long-Term Support release. Every code example in this workbook assumes .NET 10.
- **Nullable enable** — the compiler will warn you when you try to use a value that might be null without checking first. This single setting prevents the most common runtime crash in all of software: NullReferenceException.
- **ImplicitUsings enable** — common namespaces like System, System.Collections.Generic, and System.Linq are available without writing using statements at the top of every file.

Now open Program.cs. It contains one line:

```
Console.WriteLine("Hello, World!");
```

Run it:

```
dotnet run
```

What you should see:

```
Hello, World!
```

If that worked, your environment is ready. Delete the Hello, World! line — you will replace it with real TMS code in Exercise 1.

Troubleshooting:

Problem	Likely cause	Fix
dotnet --version shows 8.x or 9.x	Wrong SDK installed	Download .NET 10 from dot.net and install it
code . does nothing	VS Code not in PATH	Open VS Code manually, then File → Open Folder → select TmsCore
Nullable line is missing from csproj	Older template version	Add <Nullable>enable</Nullable> manually inside <PropertyGroup>
dotnet run says "project not found"	You are in the wrong directory	Run cd TmsCore first — you need to be inside the project folder

Exercise 1: The First Safety Net (LO 1.1: Environment + Null Safety)

The situation: The legacy TMS stored student region data as a plain string with no null checks. When the batch enrollment confirmation system tried to format mailing addresses at 2 AM, it called .ToUpper() on a region that was never assigned. The app crashed with NullReferenceException. Three hundred students received no confirmation email. Nobody noticed until Monday morning.

The fix is not to add null checks everywhere after the fact. The fix is to make the compiler tell you about the problem before the code ever runs.

Step 1 — See What the Compiler Catches

Open Program.cs. Write the following code that reproduces the legacy bug:

```
// This is how the legacy system declared region — no indication it could be empty
string region = null; // ⚠ Compiler warning CS8600
Console.WriteLine(region.ToUpper()); // ⚠ Compiler warning CS8602
```

What you should see: Two yellow squiggly underlines in your IDE, and if you run dotnet build, two warnings:

```
warning CS8600: Converting null literal or possible null value to non-nullable type.
warning CS8602: Dereference of a possibly null reference.
```

The code compiles — warnings are not errors by default — but the compiler is telling you: "This will crash at runtime." The legacy team ignored these warnings. You will not.

Step 2 — Fix It Three Ways

Replace the code in Program.cs with these three null-handling patterns. Read the comments — they explain what each operator does:

```
// Declare the variable as nullable with '?'
// This tells the compiler: "I know this might be null. I accept responsibility."
string? region = null;

// Null-conditional operator '?.' — skip the call if null
// If region is null, ToUpper() never executes. No crash.
string? upperRegion = region?.ToUpper();
Console.WriteLine($"Region (conditional): {upperRegion}");

// Null-coalescing operator '??' — provide a fallback value
// If region is null, use "Unassigned" instead.
string displayRegion = region ?? "Unassigned";
Console.WriteLine($"Region (coalesced): {displayRegion}");

// Null-coalescing assignment '??=' — assign only if currently null
// Useful for lazy initialization.
region ??= "Addis Ababa";
Console.WriteLine($"Region (assigned): {region}");
```

Run it:

```
dotnet run
```

What you should see:

```
Region (conditional):
Region (coalesced): Unassigned
Region (assigned): Addis Ababa
```

The first line is blank (not a crash) because ?. safely returned null. The second line shows the fallback. The third line shows the lazy assignment.

Step 3 — Declare Your First TMS Variables

Now write the variables that the rest of this workbook will use. Still in Program.cs:

```
string studentName = "Abeba";
string studentId = "STU-001";
int enrollmentCount = 3;
decimal grantAmount = 1999.99m;  // 'm' suffix marks a decimal literal
DateTime enrolledAt = DateTime.UtcNow;
string? campusRegion = null;

Console.WriteLine($"Student: {studentName} ({studentId})");
```

```
Console.WriteLine($"Courses: {enrollmentCount}");
Console.WriteLine($"Grant: {grantAmount:F2}");
Console.WriteLine($"Enrolled: {enrolledAt:yyyy-MM-dd}");
Console.WriteLine($"Campus: {campusRegion ?? "Not assigned"}");
```

Run it:

```
dotnet run
```

What you should see:

```
Student: Abeba (STU-001)
Courses: 3
Grant: 1999.99
Enrolled: 2026-05-09
Campus: Not assigned
```

The date will show today's date. The grant shows exactly two decimal places — no drift.

Why do professional teams rely on the CLI (`dotnet run`, `dotnet build`) instead of only clicking buttons in an IDE? Build servers and CI/CD pipelines run your code without a GUI. If your project only builds when you press F5 in Visual Studio, it cannot be deployed automatically. The CLI is the universal interface.

Troubleshooting:

Problem	Likely cause	Fix
No compiler warnings on `string region = null;`	Nullable not enabled	Check `TmsCore.csproj` for `<Nullable>enable</Nullable>`
`1999.99m` shows a red underline	Typing `1999,99m` (comma)	Use a dot for the decimal separator: `1999.99m`
`dotnet run` says "top-level statements must precede…"	You wrapped code in a class	In .NET 10 console apps, write directly in `Program.cs` — no class wrapper needed
Output says `Hello, World!` before your code	Did not delete the template line	Remove `Console.WriteLine("Hello, World!");` from the top of `Program.cs`

Exercise 2: The Ministry Audit Failure (LO 1.2: Primitives)

The situation: The Ministry of Education relies on the TMS to allocate and report on training grants. During a quarterly audit, an automated script flagged a discrepancy of 0.03 Birr — a phantom balance distributed across 100,000 student accounts. The lead architect traced it to the invoicing module's grant calculation. The legacy code used `double` for money. That single type choice created a financial error that survived three years of testing.

Step 1 — See the Bug

Open Program.cs. Add the legacy calculation:

```
// Legacy implementation — the bug that caused the audit failure
double grantPerStudent = 1999.99;
double totalAllocation = grantPerStudent * 100_000;

Console.WriteLine($"Total allocated (double): {totalAllocation}");
```

Run it:

```
dotnet run
```

What you should see: The output is close to 199999000 but includes a tiny drift — something like 199999000.00000003.

Before reading on: why did 1999.99 * 100000 produce an impossible fraction? Write your hypothesis, then compare.

The double type is a base-2 (binary) floating-point primitive. It cannot accurately represent base-10 fractional numbers like 0.99. The value 1999.99 is stored internally as an approximation. Multiply that approximation by 100,000 and the error becomes visible.

Step 2 — Fix It

Replace the calculation with the decimal type:

```
// Fixed implementation — exact financial math
decimal grantPerStudent = 1999.99m;
decimal totalAllocation = grantPerStudent * 100_000m;

Console.WriteLine($"Total allocated (decimal): {totalAllocation}");
Console.WriteLine($"Total allocated (formatted): {totalAllocation:F2}");
```

Run it:

```
dotnet run
```

What you should see:

```
Total allocated (decimal): 199999000.00
Total allocated (formatted): 199999000.00
```

Exact. No drift. The decimal type uses base-10 arithmetic internally, so 0.99 is stored as exactly 0.99.

Decision rule (use this in real projects):

- Use **decimal** when humans expect exact base-10 results (money, grants, fees, GPAs, percentages in reports).

- Use **double** when the data is inherently approximate (GPS coordinates, physics, sensor readings).

Trade-off you should know: decimal is slower than double (roughly 10x for raw arithmetic), but for business systems the correctness is worth it. If you ever see money stored as double in a code review, flag it as a bug.

Troubleshooting:

Problem	Likely cause	Fix
Compile error on 1999.99m	Using a comma: 1999,99m	Use a dot: 1999.99m
Output shows no drift with double	Display rounding hides the error	Print with :R format: $"{totalAllocation:R}" to see full precision
100_000m shows a red underline	Older C# version	The underscore separator requires C# 7+. Confirm TargetFramework is net10.0
Still using double somewhere	Mixed types in calculation	Search your method for double and replace all with decimal

Exercise 3: Pipeline Data Corruption (LO 1.3 & 1.4: Encapsulation)

The situation: In the TMS, once an Enrollment is processed, it passes through multiple middleware services logging, telemetry, notification. The database revealed that 12 students were mysteriously enrolled in NULL. The investigation found that a poorly written logging service was accidentally mutating the data during processing. Because the DTO used public set properties, the compiler could not prevent the mutation.

```
// Legacy implementation — what the logging service did to the data
public class Enrollment
{
    public string StudentId { get; set; } = string.Empty;
    public string CourseCode { get; set; } = string.Empty;
    public DateTime ProcessedAt { get; set; }
}

// Somewhere in the logging pipeline:
// enrollment.CourseCode = null;   // ← No compiler error. Data silently corrupted.
```

Step 1 — Understand Why Records Exist

If you change public `string CourseCode { get; set; }` to `public readonly string CourseCode;`, the compiler prevents reassignment after construction but using a readonly field instead of a property breaks encapsulation and prevents future validation.

The right tool is a **record**. Records are designed for immutable data once created, the values cannot be changed. They also give you value-based equality for free: two records with the same data are considered equal, unlike classes where two objects with identical data are still "different" because they occupy different memory locations.

Decision rule (use this in real projects):

- Use record for data that represents a fact after it happens — enrollment events, audit entries, grade snapshots.
- Use class for entities with lifecycle and state changes — student profiles, course configurations.

Trade-off you should know: Records reduce accidental mutation and simplify equality checks, but classes are better when identity matters (the same student object being tracked through a workflow) and when you need mutable state.

Step 2 — Create the Domain Models File

Create a file named Models.cs in your TmsCore project. This file will hold all domain types for the rest of the workbook.

Write the EnrollmentRecord as a C# record:

```
// Immutable by design — the logging pipeline cannot corrupt this
public record EnrollmentRecord(string StudentId, string CourseCode, DateTime EnrolledAt);
```

One line. The primary constructor parameters become init-only properties automatically. No one can write enrollment.CourseCode = null after construction — the compiler prevents it.

Test it in Program.cs:

```
var enrollment = new EnrollmentRecord("STU-001", "CS-401", DateTime.UtcNow);
Console.WriteLine(enrollment);

// Try to mutate it — uncomment this line and see the compiler error:
// enrollment.CourseCode = "HACKED";  // ERROR: init-only property

// Non-destructive copy — creates a NEW record with one field changed
var corrected = enrollment with { CourseCode = "CS-402" };
Console.WriteLine(corrected);

// Value equality — two records with the same data are equal
```

```
var duplicate = new EnrollmentRecord("STU-001", "CS-401", enrollment.EnrolledAt);
Console.WriteLine($"Same data? {enrollment == duplicate}");  // True
```

Run it:

```
dotnet run
```

What you should see:

```
EnrollmentRecord { StudentId = STU-001, CourseCode = CS-401, EnrolledAt = ... }
EnrollmentRecord { StudentId = STU-001, CourseCode = CS-402, EnrolledAt = ... }
Same data? True
```

The logging pipeline can no longer corrupt the data. If it needs a modified version, it must create a new record with `with { }` — the original is untouched.

Troubleshooting:

Problem	Likely cause	Fix
`EnrollmentRecord` not found in `Program.cs`	File not saved or wrong namespace	Save `Models.cs`. In a single-project console app, all files share the root namespace automatically
Compiler does not complain about `enrollment.CourseCode = "HACKED"`	You used `class` instead of `record`	Replace `public class` with `public record` and use a primary constructor
`with` expression red underline	Using it on a `class` instead of a `record`	The `with` expression only works on `record` types

Exercise 3 — Part 2: `Course` Capacity with the `field` Keyword

The situation: The `Course` entity needs to be mutable — courses can change capacity and update titles throughout a semester. But the legacy code let anyone set `Capacity = -5` with no complaint. In production, a negative capacity made the enrollment check `if (course.EnrolledCount >= course.Capacity)` pass immediately, blocking all students from a 30-seat course.

Before C# 14, enforcing validation on a property required a private backing field and seven lines of boilerplate for one simple check:

```
// Legacy Pre-C# 14 Implementation (Verbose)
public class Course
{
    private int _capacity; // Manual backing field

    public int Capacity
```

```
    {
        get => _capacity;
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException("Capacity must be positive.");
            _capacity = value;
        }
    }
}
```

C# 14 introduced the field keyword — you write validation directly in the setter and the compiler generates the backing field for you.

Add the Course class to Models.cs:

```
public class Course
{
    public required string Code { get; init; }

    public required string Title
    {
        get;
        set => field = !string.IsNullOrWhiteSpace(value)
            ? value
            : throw new ArgumentException("Title cannot be empty or whitespace.", nameof(value));
    }

    // C# 14 Auto-property validation using 'field'
    public int Capacity
    {
        get;
        set => field = value > 0
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "System constraint: Capacity must
be greater than zero.");
    }

    public int EnrolledCount { get; set; }
}
```

Test it in Program.cs:

```
var course = new Course { Code = "CS-401", Title = "Advanced C#", Capacity = 30 };
Console.WriteLine($"Course: {course.Title} (Capacity: {course.Capacity})");

// Invalid capacity — should throw
try
{
```

```
    course.Capacity = -5;
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine($"Caught: {ex.Message}");
}

// Invalid title — should throw
try
{
    course.Title = "";
}
catch (ArgumentException ex)
{
    Console.WriteLine($"Caught: {ex.Message}");
}
```

Run it:

```
dotnet run
```

What you should see:

```
Course: Advanced C# (Capacity: 30)
Caught: System constraint: Capacity must be greater than zero. (Parameter 'value')
Caught: Title cannot be empty or whitespace. (Parameter 'value')
```

The model now rejects bad data at the point of entry. No downstream service can set capacity to -5.

Decision rule (use this in real projects):

- Use `field` in properties when you need validation and still want concise code.
- Keep manual backing fields only when you need custom behavior that auto-property backing cannot express cleanly.

Trade-off you should know: `field` removes boilerplate and the compiler generates the backing field — you never see it. But be aware: property initializers (like `= "Active"`) bypass the setter and assign directly to the backing field. If you need setter validation to run on initialization, assign in the constructor instead.

Troubleshooting:

Problem	Likely cause	Fix
`field` keyword shows as red underline	Project not targeting C# 14	Confirm `<TargetFramework>net10.0</TargetFramework>` in

Problem	Likely cause	Fix
		csproj. C# 14 is the default for .NET 10
Validation never throws	You wrote public int Capacity { get; set; } without the custom setter	Replace the auto-property with the validated version shown above
required keyword error	Older C# version	required needs C# 11+. Confirm .NET 10 target

Exercise 3 — Part 3: Student Model

The enrollment pipeline in Session 2 needs a Student type with validated properties. Add this class to Models.cs now — if you skip it, Exercises 4 through 7B will not compile.

```
public class Student
{
    public required string Id { get; init; }

    public required string Name
    {
        get;
        set => field = !string.IsNullOrWhiteSpace(value)
            ? value
            : throw new ArgumentException("Name cannot be empty or whitespace.", nameof(value));
    }

    public int Age
    {
        get;
        set => field = value is >= 16 and <= 100
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "Age must be between 16 and
100.");
    }

    public decimal GPA
    {
        get;
        set => field = value is >= 0.0m and <= 4.0m
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "GPA must be between 0.0 and
4.0.");
    }
}
```

Test it in Program.cs:

```
var s = new Student { Id = "S1", Name = "Abeba", Age = 20, GPA = 3.8m };
Console.WriteLine($"Student: {s.Name}, GPA: {s.GPA}");

// These should throw — try each one:
// new Student { Id = "S2", Name = "", Age = 20, GPA = 3.0m };

// new Student { Id = "S3", Name = "Test", Age = 12, GPA = 3.0m };

// new Student { Id = "S4", Name = "Test", Age = 20, GPA = 5.0m };
```

What you should see: Student: Abeba, GPA: 3.8. Uncommenting any invalid line throws the appropriate exception with a clear message.

Exercise 3B: Interface Contract Wiring (LO 1.4: OOP Contracts)

The situation: The analytics team needs to generate grade reports for the end-of-semester review. The problem: the TMS has two completely different assessment types quizzes (graded by correct answers out of total) and lab assignments (graded by a weighted formula of functionality and code quality). The reporting pipeline needs to process both through the same method without knowing which type it is dealing with.

This is the problem interfaces solve. An interface is a contract it says "any type that implements me guarantees it can do these things." The reporting method accepts IGradable and calls CalculateGrade(). It does not care whether the object is a quiz, a lab, or something that does not exist yet.

Step 1 — Define the Contract

Add the IGradable interface to Models.cs:

```
public interface IGradable
{
    string Title { get; }
    decimal CalculateGrade();
}
```

Step 2 — Implement It on Two Assessment Types

Add both classes to Models.cs:

```
public class Quiz : IGradable
{
    public required string Title { get; init; }
    public required int CorrectAnswers { get; init; }
    public required int TotalQuestions { get; init; }
```

```
    public decimal CalculateGrade()
    {
        if (TotalQuestions == 0) return 0m;
        return (decimal)CorrectAnswers / TotalQuestions * 100m;
    }
}

public class LabAssignment : IGradable
{
    public required string Title { get; init; }
    public required decimal FunctionalityScore { get; init; }
    public required decimal CodeQualityScore { get; init; }

    public decimal CalculateGrade()
    {
        // 70% functionality, 30% code quality
        return (FunctionalityScore * 0.7m) + (CodeQualityScore * 0.3m);
    }
}
```

Step 3 — Write the Polymorphic Report

Add this to Program.cs:

```
void PrintGradeReport(IEnumerable<IGradable> assessments)
{
    Console.WriteLine("--- Grade Report ---");
    foreach (var item in assessments)
    {
        Console.WriteLine($"{item.Title}: {item.CalculateGrade():F2}%");
    }
}

// Test it — one array holds two completely different types
IGradable[] cohortAssessments = [
    new Quiz { Title = "C# Basics", CorrectAnswers = 18, TotalQuestions = 20 },
    new LabAssignment { Title = "Registration API", FunctionalityScore = 90m, CodeQualityScore = 85m }
];

PrintGradeReport(cohortAssessments);
```

Run it:

```
dotnet run
```

What you should see:

--- Grade Report ---
C# Basics: 90.00%
Registration API: 88.50%

The PrintGradeReport method processed both Quiz and LabAssignment without knowing their concrete types. When the team adds PeerReview : IGradable next semester, the report method works without a single line of change. That is the power of programming against a contract instead of a concrete class.

Troubleshooting:

Problem	Likely cause	Fix
Cannot implicitly convert type	Class does not declare it implements the interface	Ensure public class Quiz : IGradable — the : IGradable is required
Integer division — CorrectAnswers / TotalQuestions returns 0	C# integer division truncates	Cast to decimal first: (decimal)CorrectAnswers / TotalQuestions
Method does not accept the array	Parameter type mismatch	Use IEnumerable<IGradable> — arrays implement IEnumerable automatically

---

Session 1 Checkpoint

Before moving to Session 2, confirm all four:

- ☐ dotnet run produces exact decimal output for the grant calculation — no .00000003 drift
- ☐ Setting course.Capacity = -5 throws ArgumentOutOfRangeException with a clear message
- ☐ Attempting enrollment.CourseCode = "HACKED" on an EnrollmentRecord produces a compiler error (init-only)
- ☐ PrintGradeReport processes both Quiz and LabAssignment through a single IGradable parameter